Automatic Removal of Unused Code for Software Testing

Jing Lin
BSc Computer Science
Submission date: 19th January 2015
Supervisor: Jens Krinke

Acknowledgement

This project is completed with the great help of many people and I would like to give my most sincere thanks to all of them.

I would like to express my sincere gratitude and thanks to my supervisor Jens Krinke to spend his time helping me understand and giving me suggestions to carry out this project.

I would also take this chance to thank my family for their continuous support throughout my studies, my friends for their encouragement and my home university for providing the Erasmus programme to make this happen.

Abstract

The traditional testing system will report for the whole system when this fails in the test cases. However, this report could be large and specially if one needs to fix the errors of the system. This project aims to solve this problem by removing unused code automatically so that the reduced source code is focused on single test case and it will be easier to analyse and faster to test.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

During the testing of a system, it is possible to check which codes have been executed by measuring the code coverage information. There are tools that help to find out problems inside the system if a test fails. Notwithstanding, most of these tools are static analysis based, and report for the complete source code. The approach of this project intends to improve the process of code execution analysis and return a reduced source code instead of the complete one.

The reduced source code can be used for much more effective fault localisation during a test. Our proposal not only reduces the program size but also makes the testing process much easier and faster than the whole source code. The reduced system will provide the minimal source code that still produces the same results as the complete system.

1.2 Aim and goal

The aim of this project is to develop a tool called Delta Coverage which can automatically remove unused code[1] for software testing. An approach

[1]The unused code is the code that never has been executed during the testing.

called Delta Debugging, developed by Andreas Zeller[1][2], can be used for this improvement. This approach can remove parts of the source code and checks whether the reduced source code still produces the same results as the original one. For a better measurement, I focus on Delta Debugging and code coverage. And the goal of this project consists of development of Delta Coverage for C language programs.

1.3 Achievement process

To achieve all the aims and goals of this project, my contributions are the following:

1. Studying Delta Debugging's approach[1][2].
2. Studying about code coverage:
   - JaCoCo[2] is a code cover tool for Java program.
   - GCOV is a code cover tool for C program and only works on GNU Compiler Collection (GCC).
3. Studying about Delta[3], this tool uses Delta Debugging algorithm in GNU system.
4. Implementation of Delta Coverage tool with the following features:
   - Use of GCOV to cover a program code.
   - Use of Delta tool to remove as many unused code as possible.
5. Evaluation and analysis of different C programs with Delta Coverage tool.

[2]Java Code Coverage

1.4 Report overview

This project is organized in the following chapters:

Chapter 1 deals with an overview of the project, its aims, goals and achievement.

Chapter 2 details the information about Delta Debugging and its related works. It also explains, briefly, different test coverage programs such as JaCoCo and GCOV. In addition, it introduces an overview of Delta tool system information. And finally, it compares the proposed approach with dead code[3] elimination.

Chapter 3 describes thoroughly the problem statement, requirements, use cases and analysis for this approach.

Chapter 4 introduces the design of Delta Coverage tool. Also, it explains the structure and process of Delta Coverage tool implementation.

Chapter 5 describes the evaluation and analysis of the results in different test cases using Delta Coverage tool.

Chapter 6 details the conclusion of this project and possible future developments.

[3] `http://en.wikipedia.org/wiki/Dead_code`

Chapter 2

Background

2.1 Delta debugging

Delta Debugging is an algorithm that is based on hypothesis to find out bugs. It is one of the methods that can automate the scientific method of debugging. And also, this algorithm simplifies and prevents program bugs. In 1997, while Andreas Zeller was finishing his PhD thesis "Configuration Management with Version Sets[1]" at the same time he and his student started to work on the visual debugger to visualise the structures data in execution time. Then, they developed the Delta Debugging algorithm which could make debugging programs easier.

2.1.1 Simplifying and isolating failure-inducing

The research of Holger Cleve and Andreas Zeller[1] explains the utilities of Delta Debugging algorithm. They use it to simplify failing test case and isolate the difference between passing and failing test cases automatically. This algorithm classifies the outcomes of testing in three difference cases after running the algorithm that has to be debugged:

1. The result is passing (✓).
2. The result is failing (✗).

[1]https://www.st.cs.uni-saarland.de/publications/files/zeller-thesis-1997.pdf

3. The result is unsolved (**?**).

Delta Debugging would have been useful for one real case in the past decade (*Bugzilla*, the Mozilla bug database). In July 1999, there were more than 370 open bug reports that were recognised as *Bugzilla* case. To figure out the problem as soon as possible, Mozilla BugAThon [4] called for volunteers' help. They simplified input failure by turning those reports to a minimal test case. However, they wanted the bug report to be as specific as possible to help engineers to recreate the context in the failure program. And a test case as minimal as possible, which can subsume bug reports in the current and future case. At that moment, they only used binary search to isolate the problem, and had not thought about automate test cases. By using automated tests, they could simplify test cases and isolate a different failure problem automatically.

Minimizing Delta Debugging algorithm (*ddmin*) is used to simplify test cases until there is no more failure input that could be removed. The implementation is based on test for changes, we can see the details in Figure 2.1. Where $C\text{✗} = \delta_1, \delta_2, ..., \delta_n$ with all test changes, and every subset of that is a test case which can be represented with Δ_i. By definition, C✗ is considered as a minimal if it only contains one change. ∇_i is a larger number of subsets where $\nabla_i = C\text{✗} - \Delta_i$. By testing larger subsets (∇_i) of C✗, the difference is smaller as we increase the test fail chances. Notwithstanding, by testing smaller subsets (Δ_i) we can decrease these chances.

Minimizing Delta Debugging Algorithm

Let *test* and $c_{\text{✘}}$ be given such that $test(\emptyset) = \text{✔} \wedge test(c_{\text{✘}}) = \text{✘}$ hold.
The goal is to find $c'_{\text{✘}} = ddmin(c_{\text{✘}})$ such that $c'_{\text{✘}} \subseteq c_{\text{✘}}$, $test(c'_{\text{✘}}) = \text{✘}$, and $c'_{\text{✘}}$ is 1-minimal.
The *minimizing Delta Debugging algorithm* $ddmin(c)$ is

$$ddmin(c_{\text{✘}}) = ddmin_2(c_{\text{✘}}, 2) \quad \text{where}$$

$$ddmin_2(c'_{\text{✘}}, n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(\Delta_i) = \text{✘} \text{ (“reduce to subset”)} \\ ddmin_2(\nabla_i, \max(n-1, 2)) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(\nabla_i) = \text{✘} \text{ (“reduce to complement”)} \\ ddmin_2(c'_{\text{✘}}, \min(|c'_{\text{✘}}|, 2n)) & \text{else if } n < |c'_{\text{✘}}| \text{ (“increase granularity”)} \\ c'_{\text{✘}} & \text{otherwise (“done”).} \end{cases}$$

where $\nabla_i = c'_{\text{✘}} - \Delta_i$, $c'_{\text{✘}} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |c'_{\text{✘}}|/n$ holds.
The recursion invariant (and thus precondition) for $ddmin_2$ is $test(c'_{\text{✘}}) = \text{✘} \wedge n \leq |c'_{\text{✘}}|$.

Figure 2.1: Minimizing Delta Debugging algorithm[1].

Delta Debugging algorithm, *dd*, is extended from *ddmin* to isolate the failure-inducing changes between passing and failing test case. In the Figure 2.2 we can see the details about this implementation.

General Delta Debugging Algorithm

Let *test* and $c_{\text{✘}}$ be given such that $test(\emptyset) = \text{✔} \wedge test(c_{\text{✘}}) = \text{✘}$ hold.
The goal is to find $(c'_{\text{✔}}, c'_{\text{✘}}) = dd(c_{\text{✘}})$ such that $\emptyset = c_{\text{✔}} \subseteq c'_{\text{✔}} \subset c'_{\text{✘}} \subseteq c_{\text{✘}}$, $test(c'_{\text{✔}}) = \text{✔}$, $test(c'_{\text{✘}}) = \text{✘}$, and $\Delta = c'_{\text{✘}} - c'_{\text{✔}}$ is 1-minimal.
The *general Delta Debugging algorithm* $dd(c_{\text{✘}})$ is

$$dd(c_{\text{✘}}) = dd_2(\emptyset, c_{\text{✘}}, 2) \quad \text{where}$$

$$dd_2(c'_{\text{✔}}, c'_{\text{✘}}, n) = \begin{cases} dd_2(c'_{\text{✔}}, c'_{\text{✔}} \cup \Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{\text{✔}} \cup \Delta_i) = \text{✘} \text{ (“reduce to subset”)} \\ dd_2(c'_{\text{✘}} - \Delta_i, c'_{\text{✘}}, 2) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{\text{✘}} - \Delta_i) = \text{✔} \text{ (“increase to complement”)} \\ dd_2(c'_{\text{✔}} \cup \Delta_i, c'_{\text{✘}}, \max(n-1, 2)) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{\text{✔}} \cup \Delta_i) = \text{✔} \text{ (“increase to subset”)} \\ dd_2(c'_{\text{✔}}, c'_{\text{✘}} - \Delta_i, \max(n-1, 2)) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(c'_{\text{✘}} - \Delta_i) = \text{✘} \text{ (“reduce to complement”)} \\ dd_2(c'_{\text{✔}}, c'_{\text{✘}}, \min(2n, |\Delta|)) & \text{else if } n < |\Delta| \text{ (“increase granularity”)} \\ (c'_{\text{✔}}, c'_{\text{✘}}) & \text{otherwise (“done”)} \end{cases}$$

where $\Delta = c'_{\text{✘}} - c'_{\text{✔}} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |\Delta|/n$ holds.
The recursion invariant (and thus precondition) for dd_2 is $test(c'_{\text{✘}}) = \text{✘} \wedge test(c'_{\text{✔}}) = \text{✔} \wedge n \leq |\Delta|$.

Figure 2.2: Delta Debugging algorithm[1] (Where C✔means there are not test case changes).

There are some basic differences between simplification and isolation to make the failure go away. Simplification is used to remove each relevant part of the simplified test case until there are not failures anymore. And isolation is used to find one relevant part of the test case and then remove that particular part. Nevertheless, isolating is much more efficient than simplification for larger failure-inducing input case. Because, the failure produces the fail-

ure cause much faster than minimizing the test case.

To conclude, the basic idea of isolating failure-inducing changes and simplifying failed test input is to divide source changes into a set of configurations. Afterwards, apply each subset of configurations to the original program. And finally, correlate the testing result to find out the minimum faulty change set as aforementioned.

2.1.2 Locating causes of program failures

The research of Holger Cleve and Andreas Zeller[2] explains how to locate causes of program failures by using an automated test. The test results can determinate whether configuration is passing(✓), falling(✗) or non-deterministic(**?**). It can also narrow down systematically the falling and passing difference to a minimal. Their research is focused on two main areas: search in space and in time.

Searching in space is used to find the infected variables across program states. It is focused on the difference between the program states of a run whether the failure does occur or not: r✗, r✓. But, finding causes in state is not enough, because the program will pass through thousands of states from input to failure. To overcome this, Delta Debugging can be used to isolate the differences that cause the failure in one run and not in the other. That occurs because the identical input and states will produce the identical output as well. Delta Debugging can also systematically narrow down those initial difference into a small set of variables.

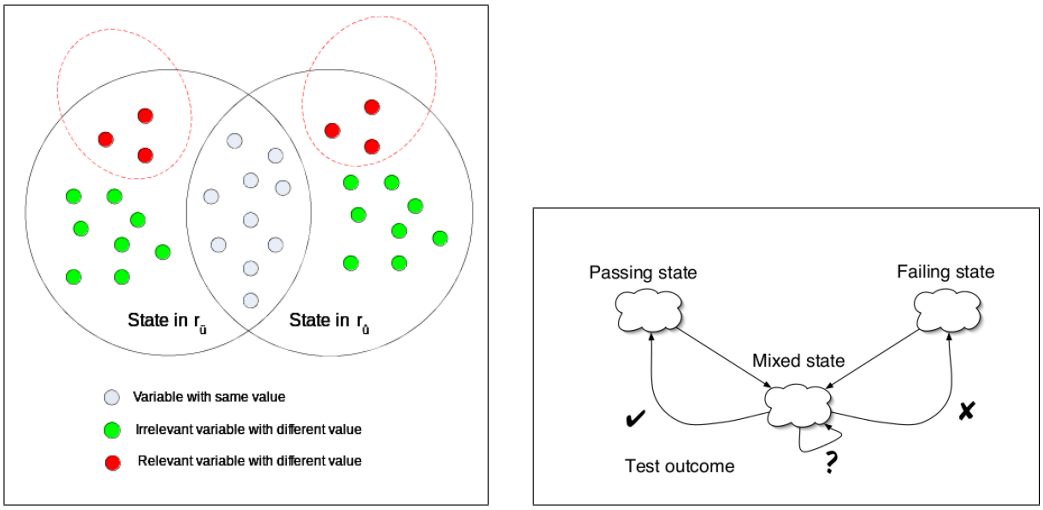


Figure 2.3: Search in space[2].

The search in space consists of comparing the program state of a passing run (r✓) and failing run (r✓) at a certain moment, see figure 2.3. However, in all the different states there are some variables that are relevant for the failure as we can see in the Figure 2.3. In this case, Delta Debugging behaviours are similar with a binary search algorithm.

Searching in time is used to search over millions of programs stated to find the moment when the defect was executed and the infection begins. It is focused on cause transitions. As a GCC example, a variable which caused the cycle even though there is no longer effect on the program failure: it is still considered as a cause transition candidate because the cycle is still there. This kind of "candidate" can be isolated automatically as the problem causes occur in the program state.

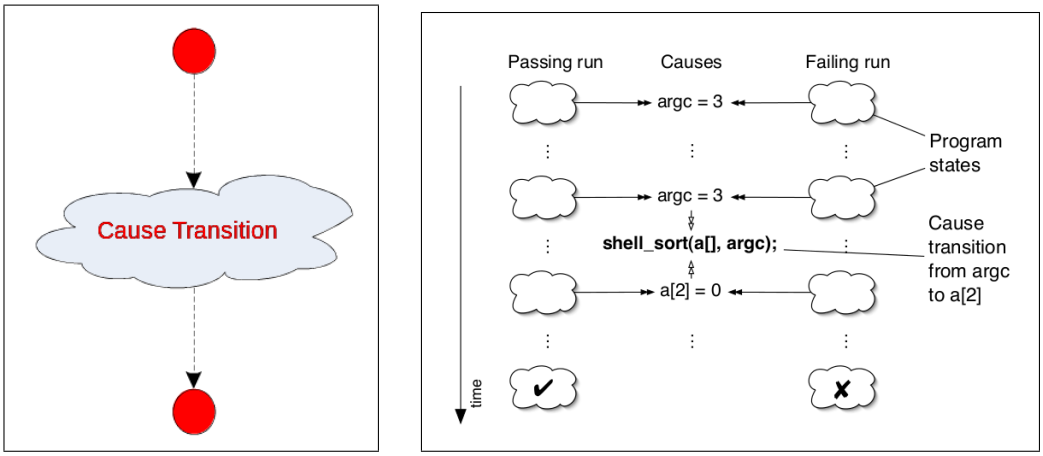


Figure 2.4: Search in time: cause transition[2].

Cause transition is where a cause is originated, it is shown on the left side of the Figure 2.4. And on the right side, we can see that the cause transition points to the program and the failure as well. During this transition, as aforementioned, there are some variables which cease to be a failure cause while the other variables begin. Because of that, we can say that cause transitions are good locations for fixes to locate the cause failure defects faster.

As an example, on the right hand, we can see how it works in the real testing case (for C program). Here, the problem is the cause transition from *argc* to $a[2]$ by calling $shell\_sort()$, where the value of $a[2]$ has changed to 0 after calling.

2.2 Code coverage

Code coverage is a measurement that is used to describe the source code degree of the program which is tested by a particular suite case. The higher code coverage not only increases the efficiency of the code, and also decreases the chance of containing software bugs[5]. Different test cases can help us to verify if the program works as expected or not. Thanks to this measurement, software developers can verify the quality of their product before they launch and offer a better final product.

In this project, we use code coverage to record the code during an execution to know which statements in a program have been executed. To measure the code coverage information, we initially focus on JaCoCo and GCOV.

2.2.1 JaCoCo

JaCoCo was developed as a replacement of EMMA[2] and Cobertura[3] to support the current Java versions. It is an Open Source toolkit to measure and report code coverage for Java program. The aim of this approach is to find out, during the software testing, which part of codes are tested by registering code executed lines[6].

Eclipse Public License is a JaCoCo distributor that offers line and branch coverage. It can instrument the bytecode while running the program. The differences between JaCoCo and the other Java coverage codes are that Clover[4] requires instrumenting the source code and Cobertura instruments the bytecode.

JaCoCo tools might not only be used to recompile the source code and add statements to the source code, but it is also used to instrument the byte code either before or while running it.

2.2.2 GCOV

GCOV is used to count how often individual program lines are executed. It is one of the test coverage program tools. GCOV has been distributed as a standard utility with the GNU Compiler Collection suite. It helps to analyse and optimise programs to discover untested parts of the program.

[2]http://emma.sourceforge.net/
[3]http://cobertura.github.io/cobertura/
[4]https://www.atlassian.com/software/clover/overview

GCOV has also been considered as a source code coverage analysis and statement-by-statement profiling tool[7]. The coverage measurement with GCOV[8] occurs with the following stages:

1. The program is set up for coverage measurement during the compilation.

2. The coverage measurement information is generated during the execution of the program.

3. The coverage information is analysed by GCOV.

Moreover, we can also use GCOV along with another profiling tool which is GPROF. GPROF can give the timing information from GCOV. It evaluates which parts of the program use the greatest amount of computing time.

By using profiler, we can find out some basic performance statistics such as:

- The frequency of execution of each line of code.
- Indicating what lines of code are executed.
- The computing time of each section of code.
- The amount of times that a statement is executed.
- Source code annotation for instrumentation.

2.3 Delta tool

Delta tool[3] is an Open Source implemented by using Delta Debugging algorithm in GNU system. Given a test shell script, it can minimize the input file content and decide if the input file is "interesting" according to the test. In this case, "interesting" refers to a file that causes a particular error as input to a program. It does not remain interesting anymore if there are no more source code or elements that could be removed.

2.4 Project approach vs. Dead code elimination

According to the compilation theory, automatic unused code removing is also known as "dead code elimination", "dead code removal" or "dead code strip"[9]. Dead code elimination is a form of compiler optimisation that removes dead code from the program.

Removing unused code has at least two benefits for software testing. On one hand, the program size is reduced, specially useful when the original program is big. On the other hand, the execution time is reduced.

Even though our project and dead code elimination reach some similar benefits, there are still differences between them. To understand it better, let us have a look at the following examples. We can see in the Figure 2.5, the local variable i is a dead variable because we never use this variable in the code fragment. The first assignation to the global variable g is also considered as a dead code. However, the third one is unreachable, because the program return g value after the second assignation. Then, it means that we can eliminate those codes.

```
int g;
void f ()
{
   int i;
   i = 1;    /* dead store */
   g = 1;    /* dead store */
   g = 2;
   return g;
   g = 3;    /* unreachable */
}
```

Figure 2.5: Dead code[9].

Unreachable codes are codes which will never be executed in the program. But, dead code is a section of source code in the program executed but never been used, which is irrelevant to the program and can be eliminated. By removing dead code, the program's output can be changed and prevent unintended bugs. After eliminating those codes, the source codes are reduced as we can see in Figure 2.6.

```
int g;
void f ()
{
   int i;
   g = 2;
   return g;
}
```

Figure 2.6: Dead code elimination[9].

However, the development of this project intends to remove unused code which has never been executed during a test. This refers not only to removing dead code and unreachable code, and also to those codes or functions which have never been used by a given test case.

Chapter 3

Requirements and analysis

3.1 Problem statement

Up to now, we can only analyse and report for the complete source code if a test fails during testing. However, we cannot analyse codes which have been executed successful and neither generate a reduced source code by removing unused code automatically.

3.2 Requirements

To solve the problem, the following requirements are needed:

- Study of Delta Debugging and automatic tests.
- Study of the code coverage: GCOV.
- Create a system that uses Delta Debugging algorithm to remove unused code.
- Prepare the developing environment which can allow to work with GNU system.

3.3 Use cases

In theory, this project has two test cases which behave as use cases.

1. By invoking GCOV, test cases are input of the selected C program. After that, the coverage information is generated.

2. By invoking Delta, the generated coverage information is considered as test cases (test script).

By executing GCOV, different coverage information will be generated with different test cases. In order to generate complete coverage information requires a set of suitable and relevant test cases.

3.4 Code coverage analysis

In order to analyse the effective of code coverage analysis, it is interesting to check how many line codes are removed by Delta Coverage tool. Indeed, it also compares the different reduced source codes in different test cases and compare them against the original source code.

3.4.1 LCOV

To better understand the code coverage analysis, we use Linux Test Project Coverage (LCOV)[10] to visualise the coverage estimation and compare the reduced source code to the original code. LCOV provides a graphical visualization of the GCOV output which can generate code coverage report as html files by using the following commands:

```
>> lcov --directory ./ --capture --output-file example.info
>> genhtml example.info
```

The '– – *directoryDIR*' is an optional flag where it can use *.da* files in DIR. The operation flag '– –*capture*' is used to capture coverage data. The

'– – *output–file FILENAME*' is an optional parameter that indicates the output filename where the execution result information should be stored in. In this case, we specify the $FILENAME$ as an *example.info*. We use '>> *genhtml example.info*' command to generate *example.info* as a html file.

3.4.2 MELD

In order to compare the reduced source code generated by different test cases, we use MELD[1] command:

```
>> meld file1 file2
```

This command will show these two files and highlight the differences. This can help us to rapidly detect the differences of these files.

[1] http://meldmerge.org/

Chapter 4

Design and implementation

4.1 Delta Coverage tool design

Delta Coverage tool originated from using Delta Debugging algorithm to minimize the program source code with coverage information of the program. The design of Delta Coverage tool is based on Delta tool and GCOV to automate the testing process by removing the unused code of the program.

4.2 Structure of Delta Coverage

The structure of Delta Coverage follows the architecture rendered in Figure 4.1 and it is organized by the following:

1. **setup.sh**, it is a script that sets up a system for coverage measurement. And, also generates the coverage information for the selected C program.

2. **test.sh**, it is a script that generates a coverage information as a test case for Delta tool.

3. **run.sh**, it is a script that generates a reduced system integrated *setup.sh* and *test.sh*.

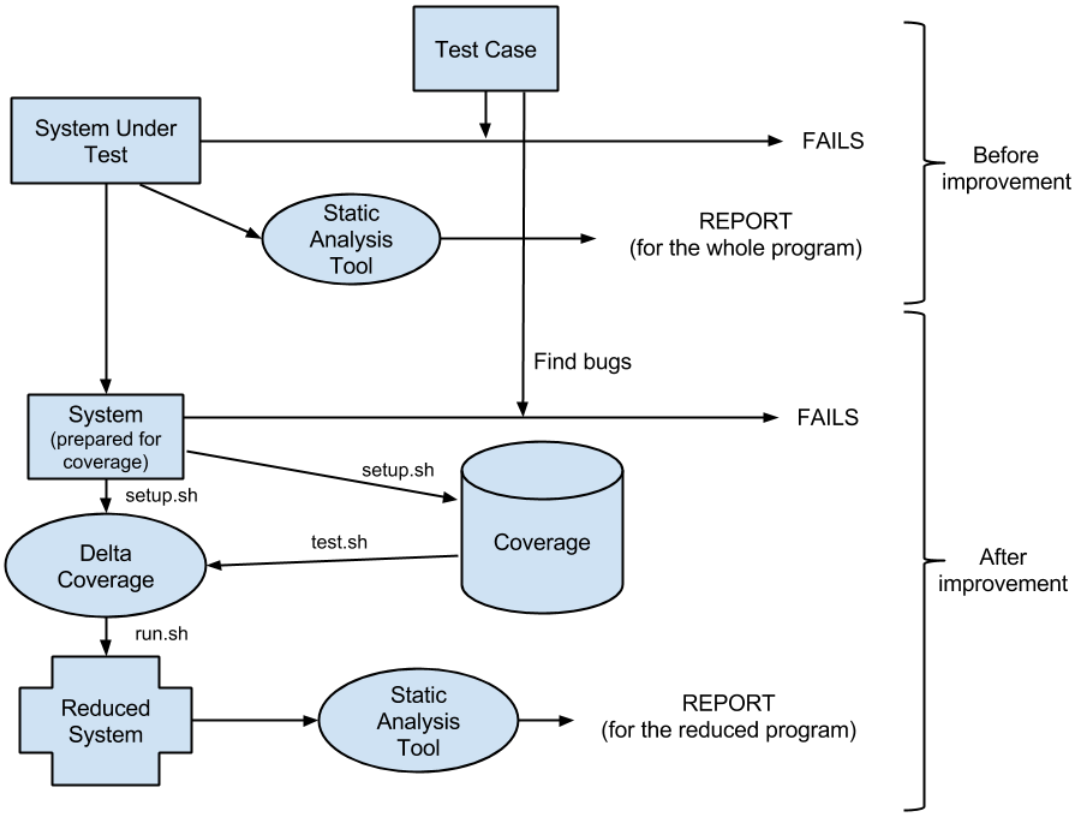


Figure 4.1: Delta Coverage architecture design.

Figure 4.1 shows the testing process improved by using Delta Coverage tool. There are two parts: before and after the improvement. Before improvement, the program is fully reported. However, after improvement, if a test fails, the program is analysed and reported for the reduced program with static analysis tool.

Figure 4.2 shows the workflow of Delta Coverage. We can see that the source code can be accepted or rejected for deletion during testing a smaller system. The accepted deletion means that the source code can be removed when it does not affect the output of the testing. However, the rejected deletion means that the source code cannot be removed when it produces different output than the original one or it cannot be compiled.

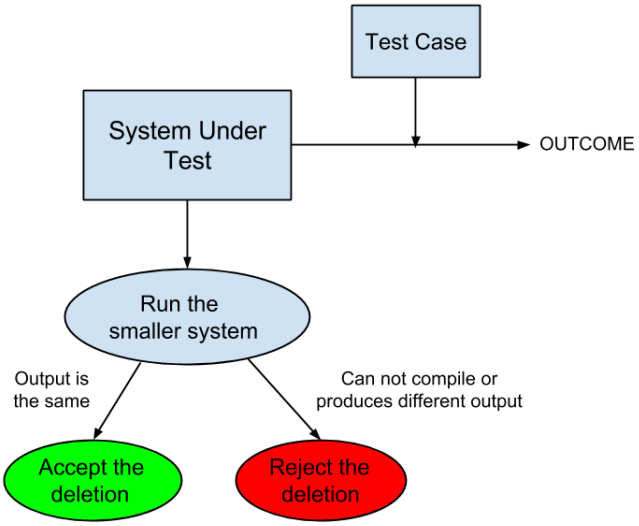


Figure 4.2: Delta Coverage workflow.

Notice that there is a difference in outcome between Delta Coverage tool and Delta Debugging (see Section 2.1.1 and 2.1.2). Delta Debugging has three outcome test cases while our approach only takes into account two of them. To be specific, Delta Coverage tool outcomes the test passing (r✓) and failing (r✗) cases, the unsolved (?) case is not considered.

4.3 Implementation of Delta Coverage

My contribution in this project is to implement the Delta Coverage for GCC compiler: *setup.sh*, *test.sh* and *run.sh*.

4.3.1 Setup.sh

During the testing of a system, *setup.sh* script is used to set up the measurement coverage system. Figure 4.3 shows the implementation of *setup.sh* (see code implementation in Appendix: C).

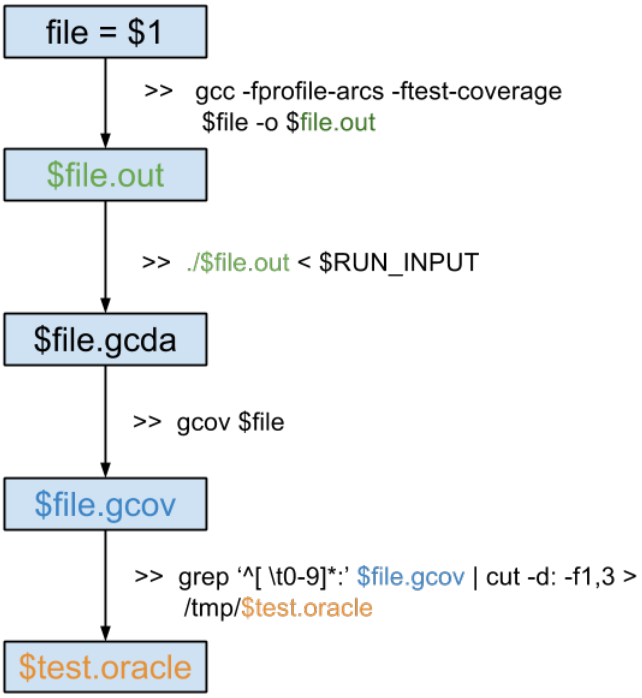


Figure 4.3: Implementation Structure : *setup.sh*

We use two special GCC options: '-fprofile-arcs -ftest-coverage'[8] to generate coverage information of the program. The '-fprofile-arcs' flag is used to instrument the code and produce the *.gcda* file. The '-ftest-coverage' flag is used to generate a *.gcno* file.

Once the program has been executed, the *.gcda* file will be created. It contains coverage information: arc transition counts, value profile counts and some other summary information.

The auxiliary file *.gcno* is created at runtime. It contains information to reconstruct the basic block graphs and its mapping to the source code. By default, *.gcno* and *.gcda* files are stored in the same directory as the object. And *.gcda* file can be stored in a separate directory by using '-fprofile-dir' flag.

According to the Figure 4.3, after the *$file* compilation with '-fprofile-arcs -ftest-coverage' special flags, an auxiliary file called $\$file.gcno$ will be created.

Then, we execute the program with a given input test file. Afterwards, the coverage information will be stored into the file $\$file.gcda$. Once we have the $\$file.gcno$ and $\$file.gcda$ files, we can now run 'gcov $file'. At this moment, the coverage information is analysed by GCOV in a given test case $\$RUN\_INPUT$. Finally, we filter the coverage information $\$file.gcov$ by:

- '$grep' \wedge [\ \backslash t 0 - 9]* :$'' to capture the lines start with numbers.
- 'cut –d: –$f 1, 3$' to cut the output with column 1 and 3.

For example, let us suppose that the test program *$file* = *toh.c*. The result of '>> gcov *$file*', after executing the previous instructions, is a file called *toh.c.gcov* and will print out the following informations:

```
File 'toh.c'
Lines executed:94.16% of 137
Creating 'toh.c.gcov'
```

Figure 4.4: The output messages of GCOV.

The messages of the Figure 4.4 indicate that the 94.16% of the whole program code has been executed except the 5.84%. The detail report is found in *toh.c.gcov* file (see Figure 4.5).

```
  -:    0:Source:toh.c
    -:    0:Graph:toh.gcno
    -:    0:Data:toh.gcda
    -:    0:Runs:1
    -:    0:Programs:1
    -:    0:Source is newer than graph
    -:    1:#include <stdio.h>
    -:    2:#include <stdlib.h>
    -:    3:#include <string.h>
    -:    4:
    ........
    .......
  560:   38:int pop(MyStack *s)
    -:   39:{
  560:   40:   int result = -1;
  560:   41:       if(s->m_data == NULL) // root node
    -:   42:       {
#####:   43:           s->m_numElements = 0;
#####:   44:           return result;
    -:   45:       }
    -:   46:       else
    -:   47:       {
  560:   48:            result = top(s);
  560:   49:            if(s->m_numElements == 1)
    -:   50:            {
    -:   51:            // last item
   64:   52:            s->m_numElements = 0;
   64:   53:            free(s->m_data);
   64:   54:               s->m_data = NULL;
    -:   55:             }
    -:   56:           else
    -:   57:           {
  496:   58:            s->m_numElements--;
  496:   59:               memmove(s->m_data, &s->m_data[1],
                       s->m_numElement s *sizeof(int));
  496:   60:              s->m_data = (int*) realloc(s->m_data,
                       s->m_numElements * sizeof(int));
    -:   61:             }
    -:   62:          }
  560:   63:     return result;
    -:   64:    }
```

Figure 4.5: A program with coverage information: *toh.c.gcov*

Figure 4.5 shows three columns:

- The first column represents the coverage information.
- The second column represents the line number of code.
- The last column is the original source code.

In the first column, the numbers indicate how often a line has been executed (e.g. line 49 is executed 560 times). The rows that start with the character '–' means that this line does not contain any code (e.g. line 39). And '#####' means that the current line code is not executed (e.g. line 43).

4.3.2 Test.sh

The implementation of *test.sh* follows the similar procedure as the *test.sh*. The only difference is that *test.sh* compares the coverage information with the one generated by *setup.sh*. We can see the difference of their structure is shown in the Figure 4.6 (see code implementation in Appendix: C).

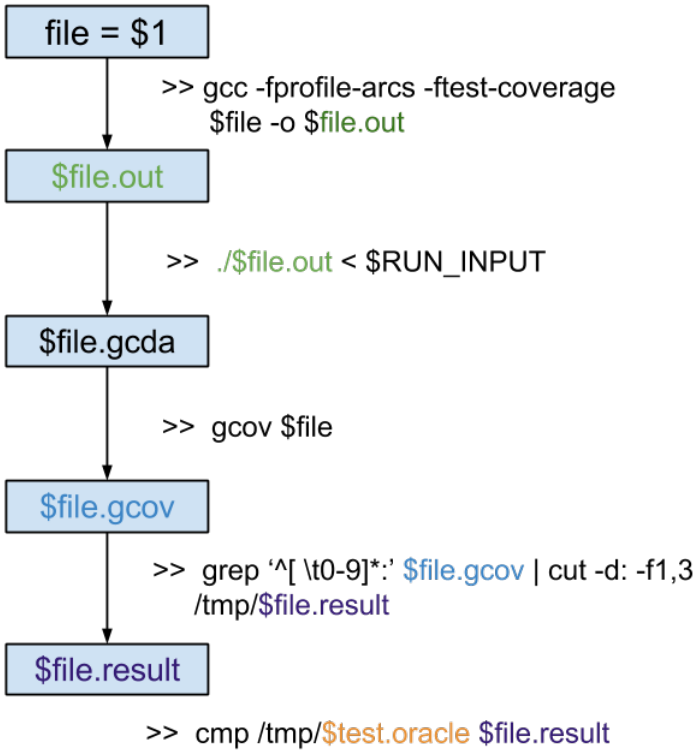


Figure 4.6: Implementation Structure: *test.sh*

4.3.3 Run.sh

The previous two scripts, actually, can be implemented in one script: *run.sh* (see code implementation in Appendix: C).

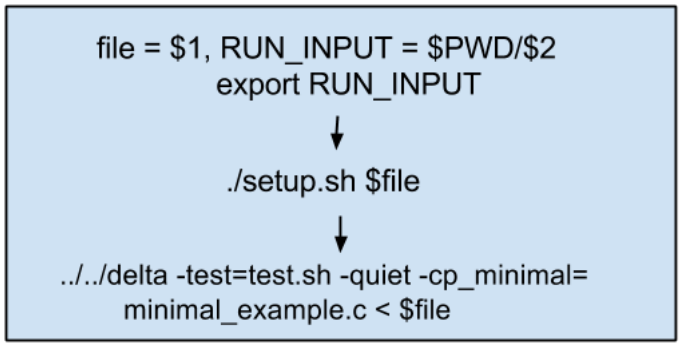


Figure 4.7: Implementation Structure: *run.sh*

This implementation takes into account the following considerations:

- The input test file RUN_INPUT should be marked as an environment variable[1] with *export* command. Otherwise, there is no way to invoke Delta with more than one given argument.
- The '–test' flag is used to specify the test script, in this case, we use the *test.sh* script before mentioned.
- The '–quiet' flag indicates no messages output. However, when this flag is disabled, we will see the verbose logging of the execution such as the granularity information related to Delta Debugging searching process.
- The '–cp_minimal' flag is used to copy the minimal successful test to the current directory.
- After invoking *run.sh*, all interesting files will be created in the *tmp∗* folder. This one contains an *arena* folder, a *log* file and *∗.c* files.
    1. The *arena* folder will contain the compilation information files, the *.gcno* file, *test.invoke* file and *∗.c* file with the generated reduced source code.
    2. The *log* file has all *∗.c* files granularity searching information and their execution time.
    3. Each *.c* file represents one Delta searching process by removing source code which has not been executed during the execution or does not affect the output result.

[1]The environment variable is set in a calling script and it is copied to the environment of the scripts it calls.

Chapter 5

Testing and result evaluation

5.1 Testing strategy

The testing strategy is based on developing relevant test cases for the selected C program. Use of relevant test case in Delta Coverage tool can generate interesting coverage information to remove a program unused code.

Different C program requires different test cases. And also, it is important the selection process of the C programs. However, to reach a relevant testing strategy, the following guidelines should be considered:

- The selected C program has been implemented with data structures algorithms as complex as possible.
- The C program should be well-known algorithm if you want to debug better.
- It would be convenient that the selected program could be executed with some input files. Otherwise, it can be fixed by changing some source codes or functions in the program.

5.2 Specific test cases

Here, we introduce a couple of test cases for the study. These test cases will be explained in the next sections.

5.3 Result evaluation based on different C programs

We have tested the Delta Coverage tool in three different C programs.

5.3.1 Tower of Hanoi Algorithm

Program function summary

Tower of Hanoi (*toh*) is a very famous game. There are three pegs and N number of disks in this game, which are placed one over the other in decreasing size. The challenge in this game consists of moving the disks one by one from the first peg to the last peg.

The selected Tower of Hanoi algorithm [1] has been implemented by using recursive logic to find the number of steps required to solve the problem. The program uses stack data structure as a peg and only costs $O(2^N - 1)$ to solve the problem. The recursive logic steps are implemented as the following general notation:
T(N, A, B, C) where T is our procedure, N is the number of disks, A is the initial peg, B is the auxiliary peg and C is the final peg.

1. T(N, A, C, B) means move top (N-1)disks from A to B.
2. T(1, A, B, C) means move 1 disk from A to C. In this case, the T procedure is the recursive base case ($N = 1$).
3. T(N-1, B, A, C) means move top (N-1) from B to C.

[1] `http://www.softwareandfinance.com/Turbo_C/TowerOfHanoi_Algorithm.html`

The input of this program could be one number or a sequence of numbers. Each number is the value of N. This program terminates when the input number is -1.

Test cases

For this program, we have created two possible relevant test cases (Figure 5.1 and 5.2) that use Delta Coverage tool to generate the reduced source code for *toh* program.

```
5
4
9
2
-1
```

Figure 5.1: Tower of Hanoi: test1.txt.

```
2
3
1
-1
```

Figure 5.2: Tower of Hanoi: test2.txt.

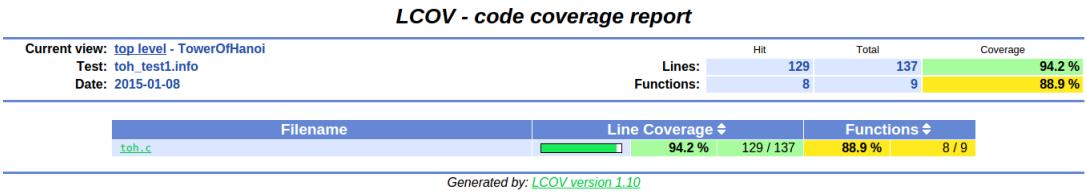
Coverage estimation

After the *run.sh* is invocated, the reduced source code is generated for each test case (*minmial_test*1.*txt_toh.c* and *minmial_test*2.*txt_toh.c*). To find out the coverage estimation of each test case we can use the following LCOV commands:

```
## To generate the code coverage report for test1.txt test case
    (similar procedure is applied to test2.txt)
>> lcov --directory . --capture --output-file toh_test1.info
```

```
## To generate the toh_test1.info as a HTML file
>> genhtml toh_test1.info
```

Afterwards, we can visualize the coverage estimation of *toh.c* for *test*1.*txt* (Figure 5.3) and *test*2.*txt* (Figure 5.4).



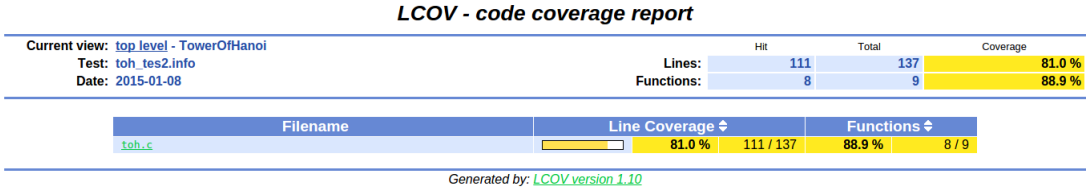
LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - TowerOfHanoi	Lines:	129	137	94.2 %
Test:	toh_test1.info	Functions:	8	9	88.9 %
Date:	2015-01-08				

Filename	Line Coverage		Functions	
toh.c	94.2 %	129 / 137	88.9 %	8 / 9

Generated by: LCOV version 1.10

Figure 5.3: Coverage estimation: *toh_test*1.*info*



LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - TowerOfHanoi	Lines:	111	137	81.0 %
Test:	toh_tes2.info	Functions:	8	9	88.9 %
Date:	2015-01-08				

Filename	Line Coverage		Functions	
toh.c	81.0 %	111 / 137	88.9 %	8 / 9

Generated by: LCOV version 1.10

Figure 5.4: Coverage estimation : *toh_test*2.*info*

From the report of these two experiments, the coverage percentage of executed functions are the same in both cases. However, the coverage percentage of executed lines are different. In the case of *test1.txt*, the coverage percentage of executed lines has a 3.2% higher than the case of *test2.txt*.

Compare the removed code with original code using LCOV

With LCOV tool, we can see the removed code. For the case *test1.txt*, in the Figure 5.5 we can see that the lines 214 and 215 (highlighted by orange color) are removed code. These two lines have never been executed because the input parameters range from 2 and 9, except -1.

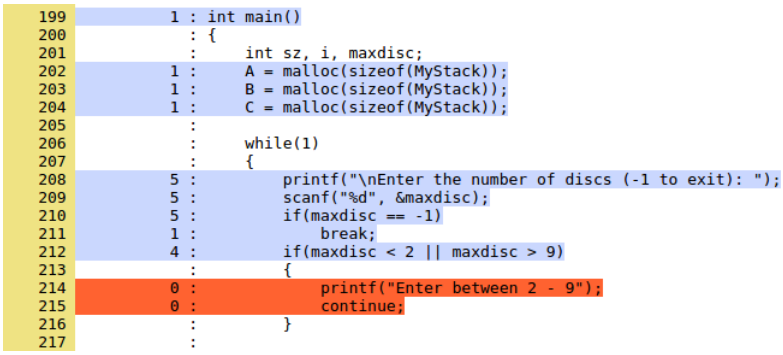


Figure 5.5: Differences between removed code and original code from code coverage report $toh\_test1.info$.

For the case of *test2.txt*, the removed covered codes are shown in the Figure 5.6 and 5.7.

Figure 5.6: Differences between removed code and original code from code coverage report $toh\_test2.info$.

For example, the lines highlighted by orange color in the Figure 5.6 have not been executed by the *test2.txt* as input parameter, because the sequence numbers of *test2.txt* have one number (2) is divisible by 2, and after executing

Solve2DiscsTOH() the variable *nDiscs* will become 1, then the *if condition* of line 156 will be accomplished, and therefore the execution will be finished.

```
184          0 :     else if (nDiscs >= 5)
185            :     {
186          0 :         SolveTOH(nDiscs - 2, source, temp, dest);
187          0 :         push(temp, pop(source));
188          0 :         movecount++;
189          0 :         PrintStacks();
190          0 :         SolveTOH(nDiscs - 2, dest, source, temp);
191          0 :         push(dest, pop(source));
192          0 :         movecount++;
193          0 :         PrintStacks();
194          0 :         SolveTOH(nDiscs - 1, temp, source, dest);
195            :     }
196          0 :     return 1;
197            : }
198            :
199          1 : int main()
200            : {
```

Figure 5.7: Differences between removed code and original code from code coverage report $toh\_test2.info$.

Another example to visualize the removed code is shown in the Figure 5.7. The sequence numbers do not have any number which is greater and equal to 5, therefore, the highlighted codes are removed.

Compare the reduced codes with Meld

We compare the two generated reduced source code with Meld. We can discover the difference percentage of coverage lines aforementioned in $test1.txt$ (Figure 5.3) and $test2.txt$ (Figure 5.4) cases. In the Figure 5.8, we can see the differences (highlighted) between these two generated codes.

- The code on the left side is the reduced source code by *test1.txt* as input.
- The code on the right side is the reduced source code by *test2.txt* as input.

Some sections on the left side are not presented on the right side. Because, the input numbers of *test1.txt* test almost all the functionalities of the original code and *test2.txt* does not. Then, the *test1.txt* is a more relevant test case than *test2.txt*. It also explains the reason of the different percentages of coverage line represented in Figure 5.3 and 5.4. However, *test2.txt* is a more relevant test case to generate a reduced code.

Figure 5.8: Differences between the reduced source codes: *minmial_test*1.*txt_toh.c* (left hand) and *minmial_test*2.*txt_toh.c* (right hand).

5.3.2 Sudoku Algorithm

Program function summary

The challenge of Sudoku game is to fulfil a grid of $9x9$ and each cell of $3x3$ with numbers from 1 to 9 and the number cannot be repeated. The selected sudoku program[2] has been implemented with backtracking to solve the given Sudoku problem.

[2]https://github.com/fxn/sudoku

Test cases

We have selected two possible relevant test cases: one is to solve an example Sudoku on Wikipedia[3] as *test*1.*txt* (see the Figure 5.9), the other one is to solve the world's hardest Sudoku introduced in 2012 by Arto InKala[4] *test*2.*txt* (see the Figure 5.10). The code has been modified for our purpose (see details in Appendix: C).

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 5.9: An example of Wikipedia Sudoku: *test*1.*txt*.

8								
		3	6					
	7			9		2		
	5				7			
				4	5	7		
			1				3	
		1					6	8
		8	5				1	
	9					4		

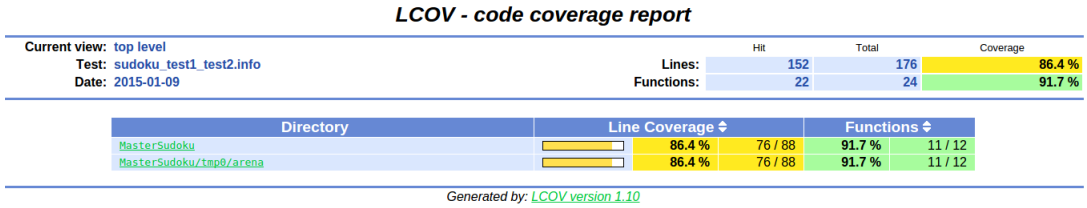
Figure 5.10: World's hardest Sudoku: *test*2.*txt*.

[3] http://en.wikipedia.org/wiki/Sudoku

[4] `http://www.efamol.com/efamol-news/news-item.php?id=43`

Coverage estimation

By executing LCOV commands aforementioned, we can see the code coverage report with $test1.txt$ and $test2.txt$ in Figure 5.11. From the report, we can see that all results are the same for both test cases.



LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level	Lines:	152	176	86.4 %
Test:	sudoku_test1_test2.info	Functions:	22	24	91.7 %
Date:	2015-01-09				

Directory	Line Coverage		Functions	
MasterSudoku	86.4 %	76 / 88	91.7 %	11 / 12
MasterSudoku/tmp0/arena	86.4 %	76 / 88	91.7 %	11 / 12

Generated by: LCOV version 1.10

Figure 5.11: Different coverage estimations with $test1.txt$ (the first row) and $test2.txt$ (the second row).

Compare the removed code with original code using LCOV

The coverage information of the removed covered code by using $test1.txt$ and $test2.txt$ are the same. By comparing the reduce code to the original one, the function $init\_known()$ has not been executed and will be deleted further (see Figure 5.12). Because the original implementation calls this function by passing the sequence input to initialise the game, but the modified version is initialised by a file ($init\_with\_file()$, line 80). The lines 93 and 94 will be deleted because the input data is correct.

```
 80          1 : void init_with_file(char *filename){
 81            :
 82          1 :   FILE *f = fopen(filename, "r");
 83          1 :   while (!feof(f)){
 84            :     char data[4];
 85         30 :     fscanf(f, "%s ", data);
 86            :     //printf("%s\n", data);
 87            :
 88            :     int i, j, n;
 89         30 :     if (sscanf(data, "%1d%1d%1d", &i, &j, &n)) {
 90         30 :       set_cell(i-1, j-1, n);
 91         30 :       known[i-1][j-1] = 1;
 92            :     } else {
 93          0 :       printf("bad input token: %s\n", data);
 94          0 :       exit(EXIT_FAILURE);
 95            :     }
 96            :   }
 97          1 :   fclose(f);
 98          1 : }
 99            :
100            : /* Processes the program arguments. Each argument is assumed to be a string
101            : with three digits row-col-number, 1-based, representing the known cells in the
102            : Sudoku. For example, "123" means there is a 3 in the cell (0, 1). */
103          0 : void init_known(size_t count, char** cells)
104            : {
105            :         int c;
106          0 :     for (c = 0; c < count; ++c) {
107          0 :         char* cell = cells[c];
108            :         int i, j, n;
109          0 :         if (sscanf(cell, "%1d%1d%1d", &i, &j, &n)) {
110          0 :             set_cell(i-1, j-1, n);
111          0 :             known[i-1][j-1] = 1;
112            :         } else {
113          0 :             printf("bad input token: %s\n", cell);
114          0 :             exit(EXIT_FAILURE);
115            :         }
116            :     }
117          0 : }
118            :
```

Figure 5.12: Differences between removed code with the original source code.

Compare the reduced codes with Meld

By comparing the two reduced source codes generated with *test*1.*txt* and *test*2.*txt*, we can see that their difference is not so relevant in Figure 5.13. Because of that, we can see the selected algorithm is optimised efficiently. Then, Delta Coverage is a good tool to determine if the testing algorithm is optimised or not.

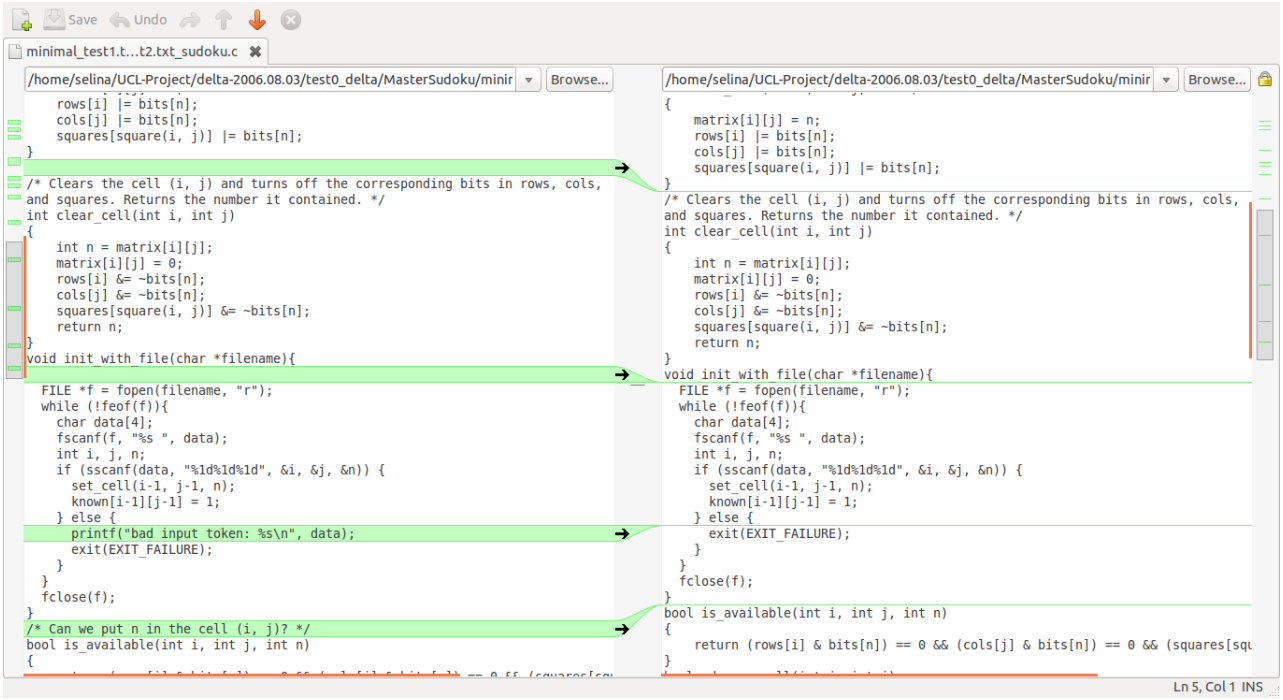


Figure 5.13: Differences between reduced source codes.

5.3.3 Travelling Salesman Problem

Program function summary

The selected program[5] is one of the classic problems that consists of a salesman who has to travel to many cities and he has to figure out which path has the lowest cost.

Test cases

We use the three test cases provided by the author of this program (for the details see Appendix: A).

Coverage estimation

In the code coverage report (Figure 5.14), the coverage estimations with *test*1.*txt* and *test*2.*txt* are the same in all the results. However, the line

[5]https://github.com/FireArrow/traveling-salesman

coverage and functions of *test*3.*txt* is 8.8% and 5.7% higher than other two test cases respectively.

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level	Lines:	465	588	79.1 %
Test:	salesma_tests.info	Functions:	36	44	81.8 %
Date:	2015-01-09				

Directory	Line Coverage		Functions	
travelling-salesman	76.4 %	155 / 203	80.0 %	12 / 15
travelling-salesman/tmp0/arena	76.4 %	155 / 203	80.0 %	12 / 15
travelling-salesman/tmp2/arena	85.2 %	155 / 182	85.7 %	12 / 14

Generated by: LCOV version 1.10

Figure 5.14: Different coverage estimations with *test*1.*txt*(the first row), *test*2.*txt* (the second row) and *test*3.*txt* (the third row).

Compare the removed code with original code using LCOV

According to the Figure 5.15, the highlighted line codes (from 316 to 318) are not executed during the test time, because the input parameters never go through it.

```
305           : /******* Input data parsing *********/
306           :
307         1 : int parseGraphFile(FILE * file) {
308           :         char cNodeA, cNodeB;
309           :         int weight;
310         1 :         int n = 0;
311           :         node * ptrA;
312           :         node * ptrB;
313           :
314        58 :         while((n = fscanf(file,"%c;%d;%c\n",&cNodeA,&weight,&cNodeB)) != EOF) {
315        56 :                 if(n != 3) {
316         0 :                         PRINT(DEBUG,"Ignoring line starting with #\n");
317         0 :                         while(getc(file) != '\n');
318         0 :                         continue; //Ignore lines starting with #
319           :                 }
320        56 :                 PRINT(DEBUG,"Pr Add: %c -> %d -> %c\n",cNodeA,weight,cNodeB);
321        56 :                 ptrA = getNode(cNodeA);
322        56 :                 ptrB = getNode(cNodeB);
323        56 :                 addEdges(ptrA, ptrB, weight);
324           :         }
325           :
326         1 :         return 0;
327           : }
328           :
```

Figure 5.15: Differences between removed code with the original source code.

Compare the reduced codes with Meld

By comparing the three reduced source codes generated with three test cases, the highlight line codes (left and right sides) represented in Figure 5.16 means that the *test*2.*txt* test case has covered less source code than the other two test cases. We can also say that the other two test cases are more complete.

In addition, the execution time of *test*3.*txt* is the highest one and *test*1.*txt* the lowest. To conclude, the complex input does not mean complete.

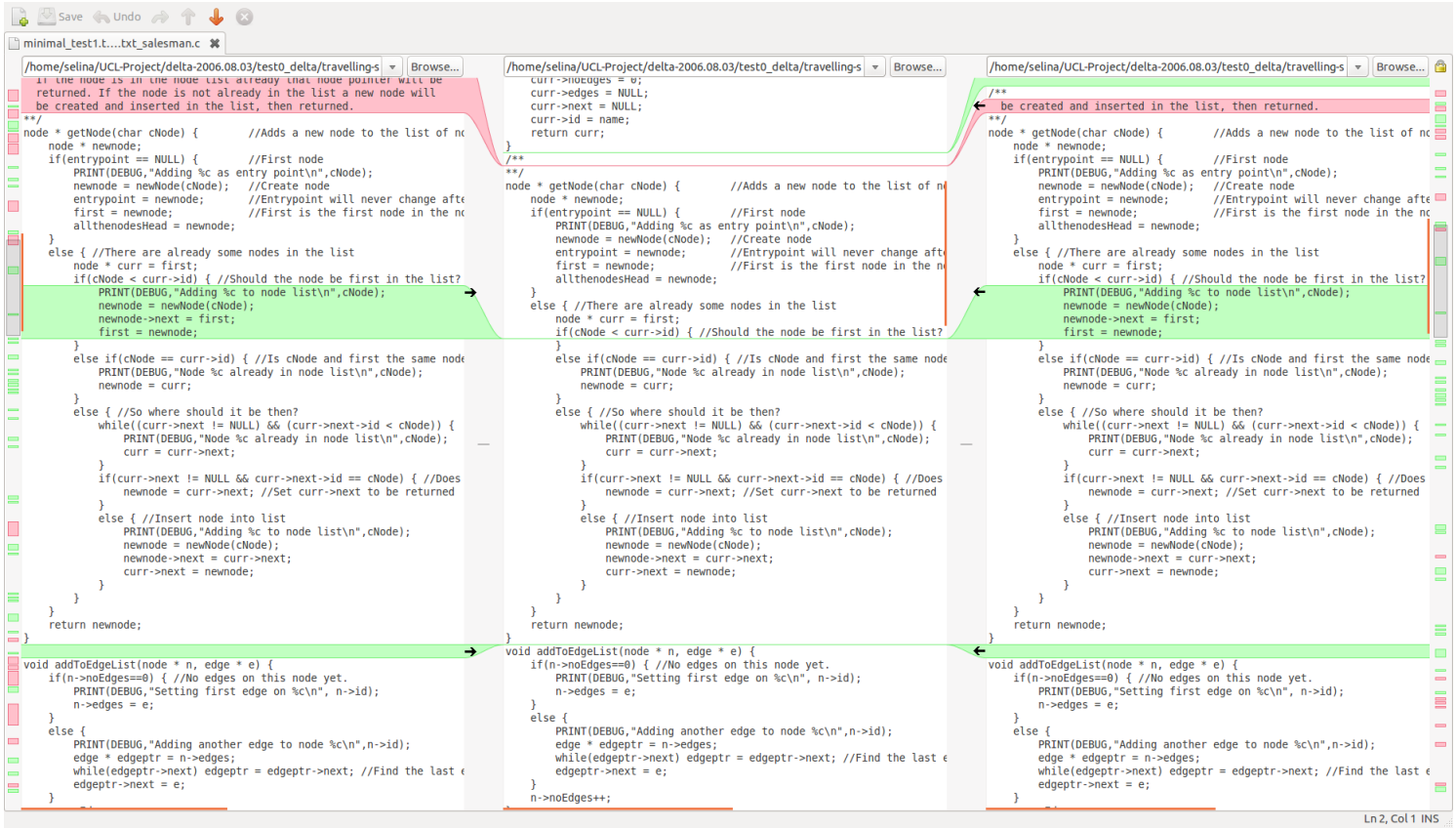


Figure 5.16: Differences between reduced source codes for the three test cases: *test*1.*txt*(left), *test*2.*txt*(centre), *test*3.*txt*(right).

Chapter 6

Conclusions

6.1 Summary

In the traditional algorithm testing system, when the algorithm fails with a test, the full system will be reported. It would be hard to figure out the errors when this report is huge. Then, we come up with the idea of Delta Coverage. This approach aims to design and implement a tool that can generate a reduced system by removing unused code automatically during the test. The design of this approach is based on Delta Debugging and measurement coverage. I have implemented two main scripts: *setup.sh* and *test.sh*. The *setup.sh* is used to set up the measurement coverage system. And *test.sh* is used to generate a coverage information that it uses as a test shell script. I have tested these implementations in three cases: Tower of Hanoi, Sudoku and Travelling Salesman algorithms. Using LCOV and MELD, I have analysed the reduced source code for different test cases and I figure out the reason of why some line codes are not executed.

6.2 Critical evaluation

In general, I think that I have achieved our aim and goal. Then, I can conclude that this project has been successfully developed. By using our approach, the process of software testing is faster than other traditional meth-

ods.

Up to now, the Delta Coverage tool works only for one file. It could be interesting to apply our approach to multiple files but this requires Multi-Delta[1] tool. The other improvement could be deleting the coverage line via TXL or Clang instead of Delta Debugging algorithm. The proposed methods are used for C program, then, it is a challenge to apply similar approaches for Java program.

6.3 Future work

In the future it would be interesting to use TXL instead of Delta Debugging. And also the improvement mentioned in the critical evaluation part.

[1]Multidelta is a wrapper that runs Delta on multiple file.

Appendix A

Evaluation data and results

A.1 Tower of Hanoi

A.1.1 Test Cases

test1.txt

```
5
4
9
2
-1
```

test2.txt

```
2
3
1
-1
```

A.2 Sudoku

A.2.1 Test Cases

test1.txt

An example of Wikipedia Sudoku.

```
115 123 157 216 241 259 265 329 338 386 418 456 493 514 548 563
    591 617 652 696 726 772 788 844 851 869 895 958 987 999
```

test2.txt

World's hardest Sudoku.

```
118 233 246 327 359 372 425 467 554 565 577 641 683 731 786 798
    838 845 881 929 974
```

A.3 Travelling Salesman Problem

A.3.1 Test Cases

test1.txt

1	A;3;B	12	E;9;B	23	G;1;B
2	B;2;A	13	E;7;C	24	G;4;C
3	B;4;C	14	E;4;D	25	G;6;D
4	C;6;A	15	E;3;F	26	G;2;E
5	C;8;B	16	F;7;A	27	G;1;F
6	C;6;D	17	F;9;B	28	G;5;H
7	D;8;A	18	F;9;C	29	H;4;A
8	D;4;B	19	F;9;D	30	H;7;B
9	D;5;C	20	F;8;E	31	H;2;C
10	D;3;E	21	F;2;G	32	H;6;D
11	E;9;A	22	G;3;A	33	H;8;E

34	H;3;F	42	I;5;F	50	J;8;E
35	H;9;G	43	I;8;G	51	J;1;F
36	H;4;I	44	I;5;H	52	J;9;G
37	I;6;A	45	I;8;J	53	J;2;H
38	I;7;B	46	J;2;A	54	J;1;I
39	I;7;C	47	J;2;B	55	J;1;K
40	I;2;D	48	J;5;C	56	A;5;K
41	I;9;E	49	J;1;D		

test2.txt

1	A;3;B	20	F;8;E	39	I;7;C
2	B;2;A	21	F;2;G	40	I;2;D
3	B;4;C	22	G;3;A	41	I;9;E
4	C;6;A	23	G;1;B	42	I;5;F
5	C;8;B	24	G;4;C	43	I;8;G
6	C;6;D	25	G;6;D	44	I;5;H
7	D;8;A	26	G;2;E	45	I;8;J
8	D;4;B	27	G;1;F	46	J;2;A
9	D;5;C	28	G;5;H	47	J;2;B
10	D;3;E	29	H;4;A	48	J;5;C
11	E;9;A	30	H;7;B	49	J;1;D
12	E;9;B	31	H;2;C	50	J;8;E
13	E;7;C	32	H;6;D	51	J;1;F
14	E;4;D	33	H;8;E	52	J;9;G
15	E;3;F	34	H;3;F	53	J;2;H
16	F;7;A	35	H;9;G	54	J;1;I
17	F;9;B	36	H;4;I	55	J;1;K
18	F;9;C	37	I;6;A	56	A;5;K
19	F;9;D	38	I;7;B	57	K;4;C

test3.txt

1	A;3;B	21	F;2;G	41	I;9;E
2	B;2;A	22	G;3;A	42	I;5;F
3	B;4;C	23	G;1;B	43	I;8;G
4	C;6;A	24	G;4;C	44	I;5;H
5	C;8;B	25	G;6;D	45	I;8;J
6	C;6;D	26	G;2;E	46	J;2;A
7	D;8;A	27	G;1;F	47	J;2;B
8	D;4;B	28	G;5;H	48	J;5;C
9	D;5;C	29	H;4;A	49	J;1;D
10	D;3;E	30	H;7;B	50	J;8;E
11	E;9;A	31	H;2;C	51	J;1;F
12	E;9;B	32	H;6;D	52	J;9;G
13	E;7;C	33	H;8;E	53	J;2;H
14	E;4;D	34	H;3;F	54	J;1;I
15	E;3;F	35	H;9;G	55	J;1;K
16	F;7;A	36	H;4;I	56	A;5;K
17	F;9;B	37	I;6;A	57	K;4;C
18	F;9;C	38	I;7;B	58	K;7;E
19	F;9;D	39	I;7;C		
20	F;8;E	40	I;2;D		

Appendix B

Project Plan

Automatic Removal of Unused Code for Software Testing

BSc Computer Science
COMP3091A: Individual Project

Deliverable: Project Plan
Authored by: Jing Lin
Student number: 14092010
Directed by: Jens Krinke
November 12th , 2014

Table of Contents

1. Introduction

The aim of this document is to describe the project plan, which includes the project aims, objectives, artefacts and work plan.

2. Project aims

The aim of this project is to develop a tool that automatically removes source code that is not executed during a test. For a better measurement I focus on delta debugging developed by Andreas Zeller [1] and automatic tests.

Delta debugging can remove parts of the source code and check if the executed tests still produce the same output as before, and find the minimal source code that still produces the same test results. We will use code coverage as test for delta debugging.

The reduced source code can then be used for much more effective fault localization or other static analysis.
For code coverage, I focus on gcov (for C program) and JaCoCo (for Java program).

3. Project objectives

3.1. Main objective

"Develop a tool which can automatically remove unused code during software testing."

3.2. General objectives

- To offer a usable and useful tool.
- To make fault localization easier.
- To develop a quality software considering the resources given.
- To reduce the program size to the relevant source code executed during a test.

4. Artefacts

4.1. Associated problems

Debugging programs without any automatic algorithm is a harder and boring job. That is why there were a lot of software developers spending a lot of time during software testing. It is because of the execution time wasted and repeated processes with wrong results. Consequently, software developers may think about the improvement they can make. To solve that problem a static analysis tool has been developed.

Delta debugging is one of the approaches which can automate the scientific method of debugging and it was developed by Andreas Zeller. This algorithm can simplify and prevent problems. In 1997, while Andreas Zeller finished the version control of his PhD thesis "Configuration Management with version sets" [1], he was thinking about doing some useful thing for change. Because of that, he was searching for code history with one of his students. They started to come up with the visual debugger call to visualize the data structures, and then, they developed the algorithm to make debugging programs easier.

Thanks to Andreas Zeller and other static analysis tool developers, today everyone can use those for free and save time. The only one inconvenience is that we can only report the whole program with those tools and an approach. That is the main reason of this project research: how to report the program without unused codes?

Due to the above discussions, as a result, the idea is to use delta debugging and code coverage to develop our tool which can automatically remove unused code.

4.2. Delta Debugging

Delta debugging is an algorithm based on hypothesis to find bugs. The idea is to have an automated test which can give you whether configuration is falling, passing or whether it is nondeterministic. Using automatic strategy in delta debugging can systematically narrow down the difference between the falling and passing run in the program, until it has never given you the difference to a minimal difference [2].

It is also use to simplify and isolate failure-inducing circumstances automatically, such as the program input, program executions, or changes to the program code. To understand it better, let's analyse the Bugzilla case happened in July 1999. Since the Mozilla bug database had more than 370 open bug reports at that moment, the Mozilla BugAThon [3] called for volunteers to help simplifying. In this case, simplifying means turning these bug reports to minimal test cases, where every part of the input would be

significant in reproducing the failure. Suppose that you have a big sequel query, for instance, and after you send this to a database, the database crashes. Then, to understand which part of the input made the program fail, we should test for change using delta debugging algorithm. It means changing what causes a failure of the input, then, decomposing changes, and finally, doing the test cases and tests.

Using all the above notions, we can identify and break the cause-effect chain during a software testing.

4.3. Code coverage

Code coverage is a measure used to describe the degree to which the source code of a program is tested by a particular suite. - High code coverage increases the efficiency of the code, and decreases the chance of containing software bugs [4]. Software developers use coverage testing to make sure that software is actually good enough for a release. Test suites can verify that a program works as expected. By using coverage program tests we can know how much of the program is exercised by the test suite. Developers can determine what kinds of test cases need to have a better testing and a better final product.

This project will use code coverage to record the code during an execution which statements in a program have been executed. - To measure the code coverage we have to instrument codes from the program and I will use the JaCoCo for Java program and gcov for C program.

4.3.1. JaCoCo

JaCoCo means “Java Code Coverage”, it is an Open Source toolkit for measuring and reporting for Java programs. The aim is to find out during the software testing which parts of code are tested by registering the lines of code executed [5].

JaCoCo tools are distinguished in two main kinds: the first tool is required to recompile the source code and add statements to the source code; the second tool is for instrumenting the byte code either before or while running it.

JaCoCo is distributed under the terms of the Eclipse Public License, which offers line and branch coverage and instrument the byte code while running the code. The differences between JaCoCo and the other Java coverage codes are that Clover requires instrumenting the source code and Cobertura instruments the byte code offline [6].

4.3.2. Gcov

Gcov is a test coverage program, distributed as a standard utility with the GNU Compiler Collection suite. Using gcov can help you to analyse your program more efficiently, run your codes faster and discover untested parts of the program. Gcov is also a source code coverage analysis and statement-by-statement profiling tool [7] to discover optimization efforts in the program. We can also use gcov along with another profiling tool, gprof, to assess which parts of the program use the greatest amount of computing time.

Using a profiler such as gcov or gprof, we can find out some basic performance statistics, such as:

- How often each line of code executes?
- What lines of code are actually executed?
- How much computing time each section of code uses?

It can count the amount of times each statement in a program is executed and annotate source code to add instrumentation.

4.4. New solution

To figure out software testing needs, given a whole program or system under test, we have to instrument the code. Then, we have to use a system prepared for coverage to cover a code and test case to finding bugs. After that, we will use our tool to produce a reduced system based on instrumented code and coverage result. Finally, we can report for the reduced program using a static analysis tools.

Our tool development is based on removing fragments given a whole program or system under software testing. Then, it is based on running it in the smaller system, and repeating the process independently on the deletion results until there are no more fragments which can be removed. We can know if the fragment can be deleted or not when it runs in the smaller system. On the one hand, we accept the deletion if the output is the same, which means, this part of the fragments does not affect the program results. On the other hand, we reject the deletion if they cannot compile or produce a different output.

5. Work plan

5.1. Schedule

5.1.1. Estimated project duration

The estimated project duration is approximately 4 months because of the Erasmus program. The project starts on September 11th 2014 and the deadline is on January 14th 2015.

5.1.2. Considerations

It is important to consider that the initial planning can be revised and updated because of the evolution of the project. The planning will depend on two possible methodologies, which are delta debugging and automatic test tools such as JaCoCo and gcov, which can appear as new requirements and alter the proposed planning. If there is enough time, I may probably replace the delta debugging with TXL, because TXL is a unique programming language specifically designed to support computer software analysis and source transformation tasks.

Furthermore, we have to consider that the project must be developed more or less around the 12th of December, because it is the last official week course in my host university before winter holidays. According to that, I will be using the fourth month to evaluate the development and to finish the project writing tasks.

5.2. Project phases

The project technical phases will be based on tasks, which means that to continue the next phase we have to finish the previous task. And it will have the following stages:

5.2.1. September to October (3,5 weeks): Literature search and review.

This stage consists of developing a planning, feasibility study, analysis and design to develop our tool which can automatically remove unused code. - I started to read and study about delta debugging and code coverage.

In this stage I also did the initial set up which is oriented to prepare the environment and create the necessary systems to develop a new tool.

The following hardware and software resources are needed:

Hardware:

- Hp Pavilion dv6

Software:

- Ubuntu 14.04
- Delta.Stage.Tigris.org (Open Source Software Engineering Tools)
- Eclipse

5.2.2. October to mid-November (4,5 weeks): Tool development

In this phase we are going to analyse and study the best solution to develop our tool in two steps.

The first one, is to figure out how to instrument Java Code and C code to collect coverage information, which includes a full background study on code coverage collection. I am focusing on JaCoCo and gcov to achieve all the requirements in this stage.

The second step is to figure out how certain code elements can be removed while the rest of the program still compiles and runs. This means automatically removing codes that have not been covered by the previous step.

The main idea is to cover only code statements of a program, which have been executed during its runtime. The gathered data is then used to find and lines have not been executed which are then to be deleted from the source file. There are multiple ways to approach this:

i. Capture the coverage with gcov for C programs and then delete the lines via TXL or Clang.
ii. Instrument the program via TXL or Clang to capture coverage data (similar to i), but replacing gcov).
iii. Capture the coverage with some tool for Java programs and the delete lines via TXL or some other Java-based tool.
iv. Instrument the Java program similar to ii.

I will probably research on the four different approaches and do a comparison of the advantages and disadvantages of them. I am going to use Java, C, C++ as language to be analysed.

5.2.3. Mid-November to December (4 weeks): Tool integration

This stage consists of linking the different analysis tested as we have seen in the previous phase. And then, according to the results, decide which methodologies are better to achieve our solution.

5.2.4. December to mid-December (2 weeks): Tool testing and evaluation

This stage consists of validating the new tool which I developed, and test a system to prove that our tool can remove the unused code automatically. It will compare what has been removed to what has been covered during execution. The evaluation will be based on finding bugs, PMD and then compare the results of the original system and the reduced system using static analysis tool.

5.2.5. Mid-December to mid-January (4 weeks): Work on final report

The final stage consists of closing the project development definitively. The final report will be provided.

The next software is needed to give a good documentation and close the project:

- Microsoft Office 2013
- Adobe Reader XI

5.2.6. Other tasks during the project

1) Documentation
   Every aforementioned project phase includes a documentation phase. The outcome of this phase is a document summarizing the results of the particular project phase and giving important background information.

   To ensure that we can view the document everywhere by network connection and good documentation, the next software and app will be needed:
   - Microsoft Office 2013
   - Google Drive

2) Control meetings
   This phase will be present in every phase as we will have a weekly meeting with my project tutor because of the tight timetable. Every meeting will take around one hour, to ensure and supervise that each weekly task has been developed correctly and successfully.

   The next app will be needed for any unexpected cases about weekly meeting time:

   - UCL outlook

3) Bureaucratic tasks
   This phase is the most important part of this project, because it will guarantee all my development is correctly for the presentation day. I will send my final

report to the director to review and to get his approval, to ensure that this phase will be processed correctly by the tight timetable. We will also discuss about those task reports during the weekly meeting section.

We will use this app as an official communication in my host university:

- UCL outlook

6. Bibliography

[1] Andreas Zeller. (2002). Simplifying and isolating failure-inducing. IEE transactions on software engineering. 28 (2), p1-15.

[2] Andreas Zeller, Holger Cleve. (2005). Locating causes of program failures. ICSE'05. - (-), p1-10.

[3] Gerv (indivual mozilla.org). (2006). *What is the BugAThon?.* Available: http://www-archive.mozilla.org/newlayout/bugathon.html. Last accessed 7th Nov 2014.

[4] Joan C. Miller, Clifford J. Maloney (2012). *Systematic mistake analysis of digital computer programs*. 2nd ed. New York: ACM. p58-59.

[5] Geertjan Wielenga (2013). Code Coverage for Maven Integrated in NetBeans IDE 7.2. Oracle Coporation: Retrieved 3. 12th Oct 2014.

[6] Glenford J. Myers (2004). *The Art of Software Testing*. 2nd ed. Wiley: ISBN 0-471-46912-2. p15.

[7] Brian J. Gough. (2012). An Introduction to GCC - for the GNU compilers gcc and g++ . *Coverage testing with gcov*. 10 (-), p8.

Appendix C

Code Listing

C.1 Delta Coverage tool implementation source code

C.1.1 setup.sh

The implementation of setup.sh with shell script.

```
#!/bin/bash
# -*-sh-*-

echo $@ > test.invoke
f=$1

if gcc -fprofile-arcs -ftest-coverage $f -o $f.out > cmp_out 2>
    cmp_err; then
     if ./$f.out < $RUN_INPUT > run_out 2> run_err; then
            if gcov $f > gcov_out 2> gcov_err;then
                   grep '^[ \t0-9]*:' $f.gcov | cut -d: -f1,3 >
                       /tmp/test.oracle
            exit $?
            fi
     fi
```

```
fi
exit 1;                         # Failure
```

C.1.2 test.sh

The implementation of test.sh with shell script.

```
#!/bin/bash
# -*-sh-*-

set -x
(
echo $@ > test.invoke
f=$1

if gcc -fprofile-arcs -ftest-coverage $f -o $f.out > cmp_out 2>
    cmp_err; then
     if timeout 1s ./$f.out < $RUN_INPUT > run_out 2> run_err;
        then
            if gcov $f > gcov_out 2> gcov_err;then
                   grep '^[ \t0-9]*:' $f.gcov | cut -d: -f1,3 >
                       $f.result
            fi
     fi
     cmp /tmp/test.oracle $f.result
     exit $?
fi
exit 1;                         # Failure.
) > $f.log 2>&1
```

C.1.3 run.sh

The implementation of run.sh with shell script.

```
#!/bin/sh
```

```
# -*-sh-*-

echo $@ > test.invoke
f=$1
RUN_INPUT=$PWD/$2
minimal="minimal_"$2"_"$f
export RUN_INPUT

./setup.sh $f
../../delta -test=test.sh -quiet -cp_minimal=$minimal < $f
```

C.2 The addition function for Sudoku program

The source code of the selected Sudoku algorithm[1] has been modified for our purpose. By adding *init_with_file*() function to initialised input by file instead of passing sequence by calling *init_know*() function.

```
void init_with_file(char *filename){

    FILE *f = fopen(filename, "r");
    while (!feof(f)){
     char data[4];
     fscanf(f, "%s ", data);
     //printf("%s\n", data);

     int i, j, n;
     if (sscanf(data, "%1d%1d%1d", &i, &j, &n)) {
        set_cell(i-1, j-1, n);
        known[i-1][j-1] = 1;
     }
     else {
```

[1] https://github.com/fxn/sudoku

```
            printf("bad input token: %s\n", data);
            exit(EXIT_FAILURE);
         }
        }
        fclose(f);
}
```

Bibliography

[1] Andreas Zeller. (2002). Simplifying and isolating failure-inducing. IEE transations on software engineering. 28 (2), p1-15.

[2] Andreas Zeller, Holger Cleve. (2005). Locating Causes of Program Failures. 27th International Conference on Software Engineering. - (-), p1-10.

[3] Delta Tigris.org. (2006). Delta Tigris. Available: `http://delta.stage.tigris.org/using_delta.html`. Last accessed 4th Jan 2015.

[4] Gerv(individual mozilla.org).(2006). What is the BugAThon?. Available: http://www-archive.mozilla.org/newlayout/bugathon.html. Last accessed 7th Nov 2014.

[5] Joan C. Miller, Clifford J. Maloney.(2012). Systematic mistake analysis of digital computer programs. 2nd ed. New York: ACM. p58-59.

[6] Mountainminds GmbH & Co. KG and Contributors. (2015). JaCoCo Mission. Available: http://www.eclemma.org/jacoco/trunk/doc/mission.html. Last accessed 15th Nov 2014.

[7] Brian J. Gough. (2012). An Introduction to GCC - for the GNU compilers gcc and g++. Coverage testing with gcov. 10 (-), p8.

[8] John Wiley & Sons Ltd. (-). Profiling using GPROF and GCOV. Profiling using GPROF and GCOV. 22196 (16), 245-254.

[9] Nullstone Corporation. (2011-2012). Dead Code Elimination. Available: http://www.compileroptimizations.com/category/dead_code_elimination.htm. Last accessed 8th Oct 2014.

[10] Embeddedlnn. (2013). Test Coverage analysis with GCOV. Available: https://embeddedinn.wordpress.com/tutorials/test-coverage-analysis-with-gcov/. Last accessed 29th Dec 2014.